An Introduction to Dynamic Symbolic Execution and the KLEE Infrastructure

Cristian Cadar

Department of Computing

Imperial College London

SOFTWARE RELIABILITY GROUP

Imperial College London

International Conference on Software Testing and Analysis (ISSTA'16)

Saarbrücken, Germany, 19 July 2016

Dynamic Symbolic Execution

- Dynamic symbolic execution is a technique for ***automatically exploring paths*** through a program
  - Determines the feasibility of each explored path using a ***constraint solver***
  - Checks if there are ***any*** values that can cause an error on each explored path
  - For each path, can generate a ***concrete input triggering the path***

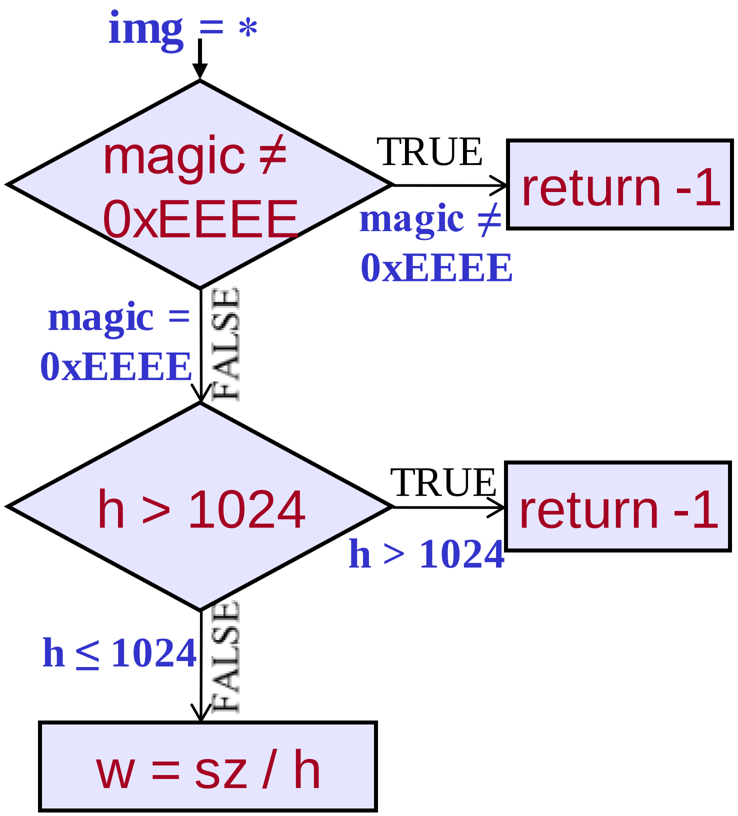
Dynamic Symbolic Execution

- Received significant interest in the last few years
- Many dynamic symbolic execution/concolic tools available as open-source:
  - **CREST, KLEE, SYMBOLIC JPF**, etc.
- Started to be adopted/tried out in the industry:
  - Microsoft (**SAGE, PEX**)
  - NASA (**SYMBOLIC JPF, KLEE**)
  - Fujitsu (**SYMBOLIC JPF, KLEE/KLOVER**)
  - IBM (**APOLLO**)
  - etc.

Toy Example

```
struct image_t {
     unsigned short magic;
     unsigned short h, sz;
     ...
```

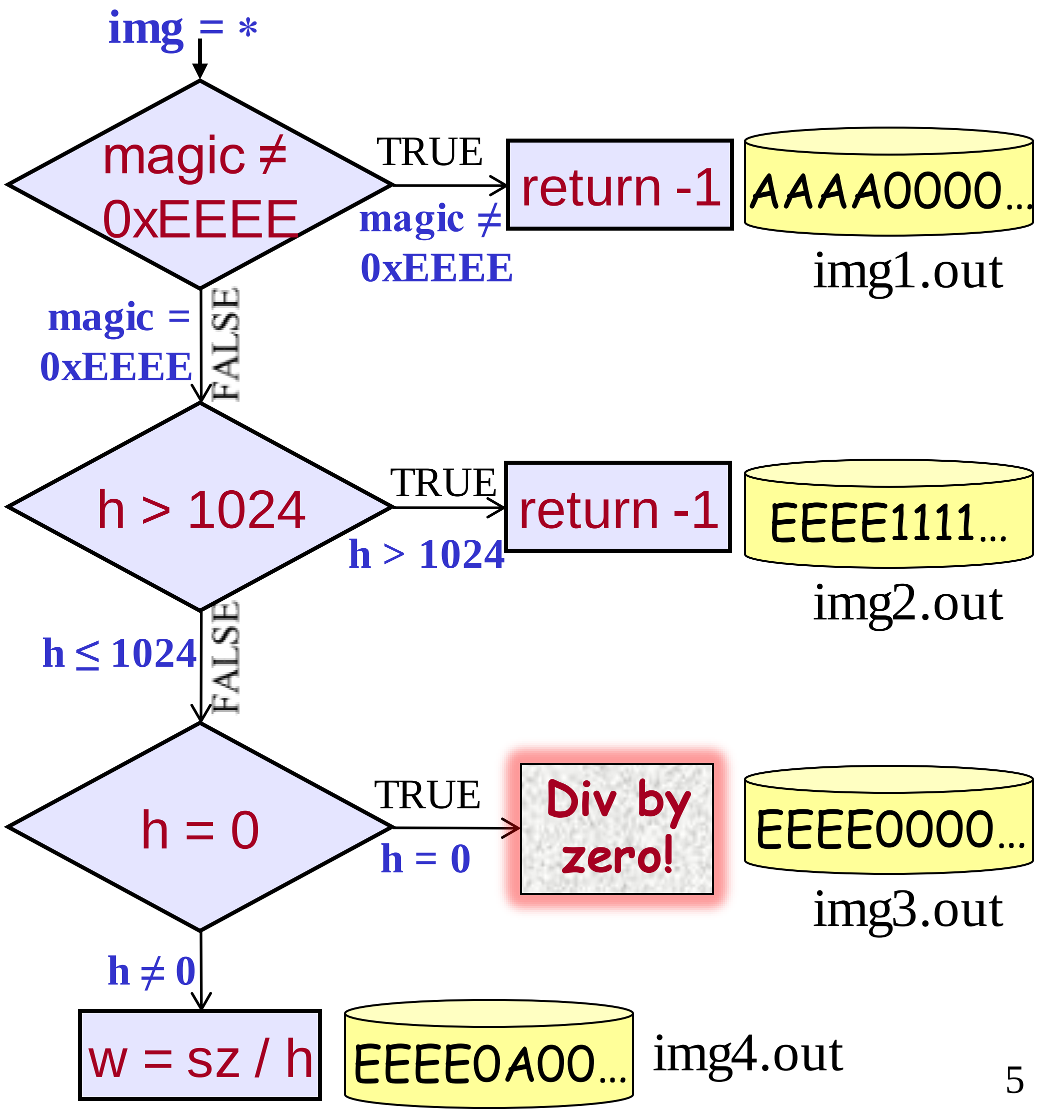
```
int main(int argc, char** argv) {
  ...
  image_t img = read_img(file);
  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  w = img.sz / img.h;
  ...
}
```



Toy Example

```
struct image_t {
    unsigned short magic;
    unsigned short h, sz;
    ...
```

```
int main(int argc, char** argv) {
  ...
  image_t img = read_img(file);
  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  w = img.sz / img.h;
  ...
}
```



All-Value Checks

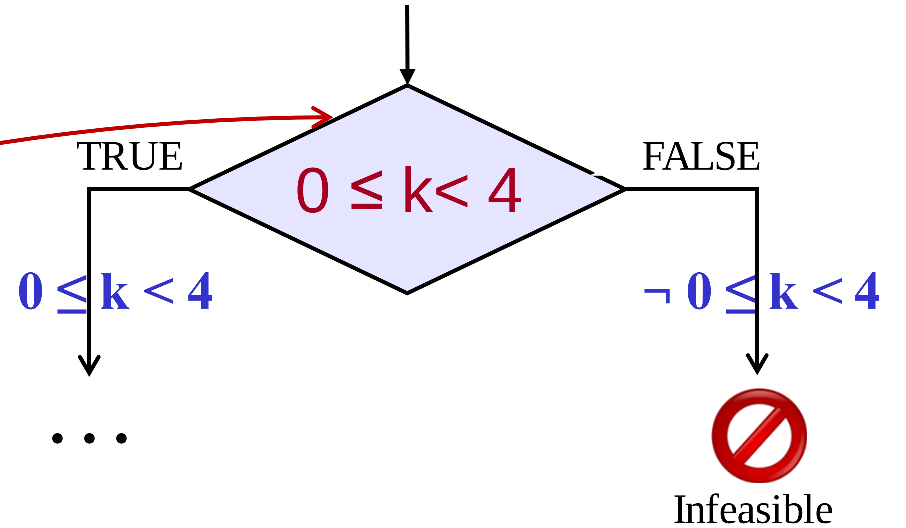
Implicit checks before each dangerous operation

- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

All-value checks!

- Errors are found if **any** buggy values exist on that path!

```
int foo(unsigned k) {
  int a[4] = {3, 1, 0, 4};
  k = k % 4;
  return a[a[k]];
}
```



All-Value Checks

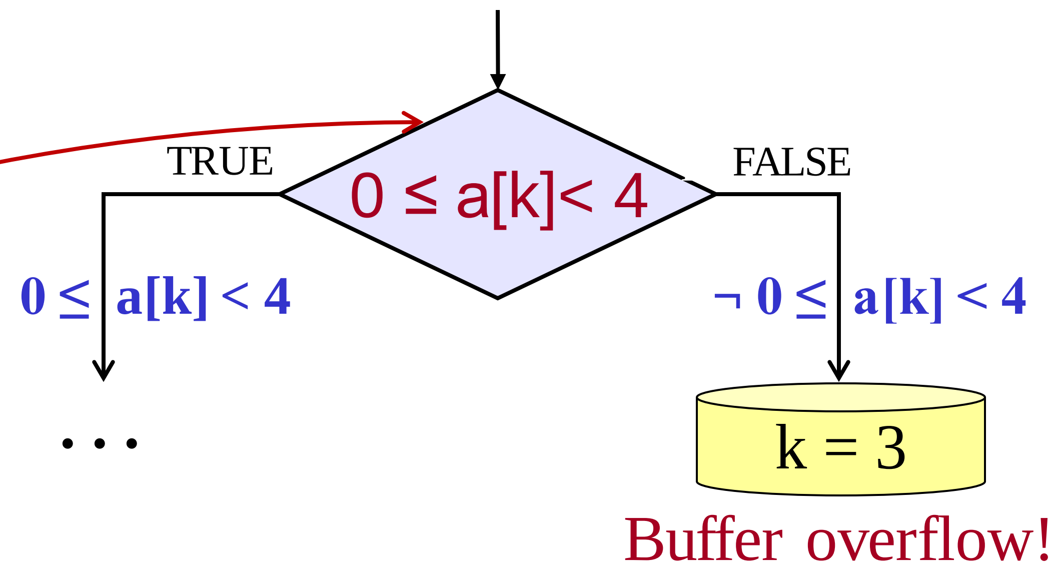
Implicit checks before each dangerous operation

- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

All-value checks!

- Errors are found if **any** buggy values exist on that path!

```
int foo(unsigned k) {
  int a[4] = {3, 1, 0, 4};
  k = k % 4;
  return a[a[k]];
}
```



Mixed Concrete/Symbolic Execution

All operations that do not depend on the symbolic inputs are (essentially) executed as in the original code

Advantages:

- Ability to interact with the outside environment
  - E.g., system calls, uninstrumented libraries
- Can partly deal with limitations of constraint solvers
  - E.g., unsupported theories
- Only relevant code executed symbolically
  - Without the need to extract it explicitly

KLEE

- Symbolic execution tool started as a successor to EXE
- Based on the LLVM compiler, primarily targeting C code
- Open-sourced in June 2009, now available on GitHub
- Active user base with over 300 subscribers on the mailing list and over 35 contributors listed on GitHub

Webpage: **klee.github.io**

Code: **https://github.com/klee**

KLEE

- Extensible platform, used and extended by many groups in academia and industry, in the areas such as:
  - bug finding
  - high-coverage test input generation
  - exploit generation
  - automated debugging
  - wireless sensor networks/distributed systems
  - schedule memoization in multithreaded code
  - client-behavior verification in online gaming
  - GPU testing and verification, etc.

An incomplete list of publications and extensions available at:

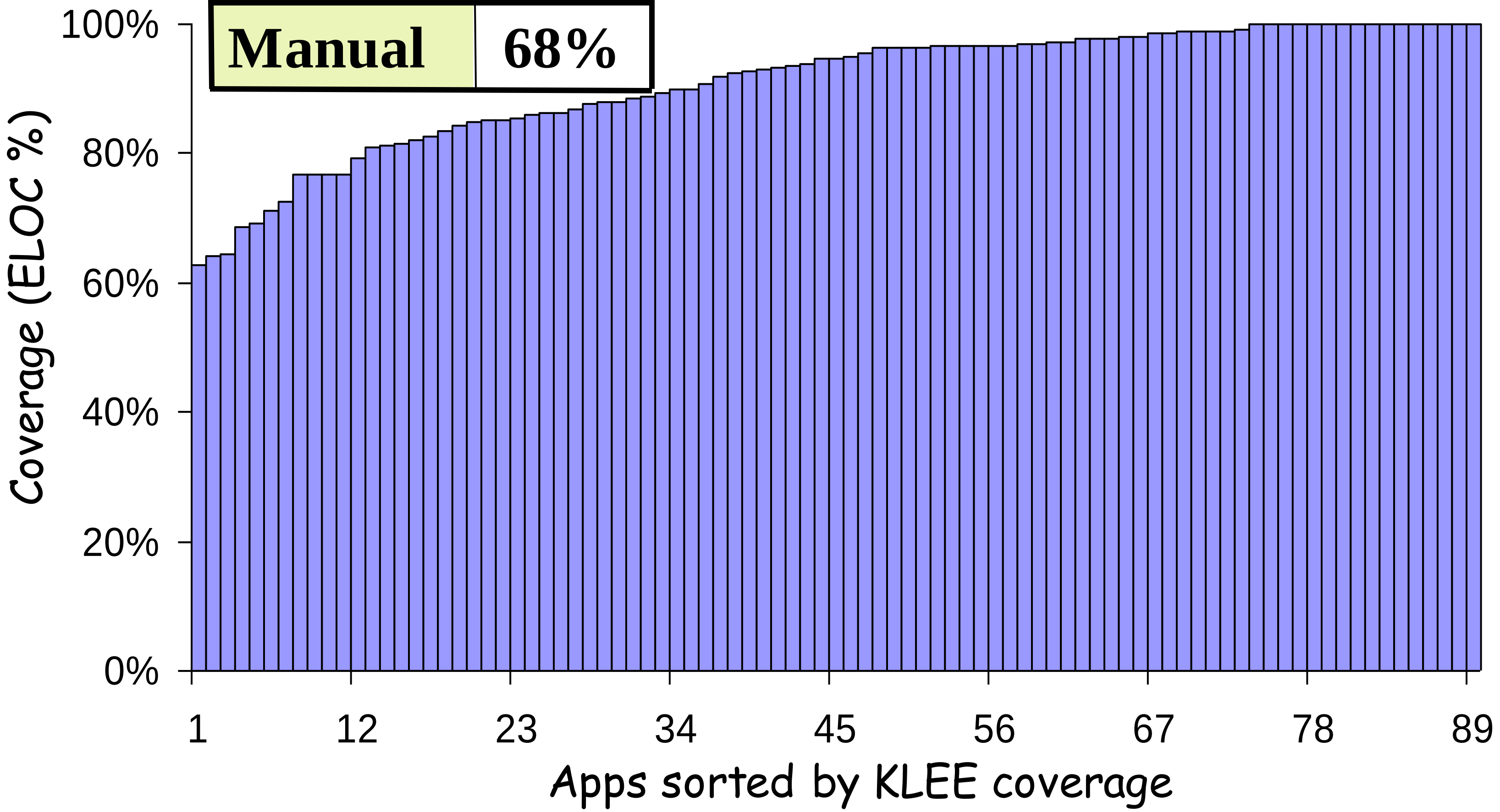
klee.github.io/Publications.html

High Line Coverage

(Coreutils, non-lib, 1h/utility = 89 h)

Avg/utility

KLEE	**91%**
Manual	**68%**



[Cadar, Dunbar, Engler OSDI 2008]

Bug Finding with KLEE (incl. EGT/EXE):
Focus on Systems and Security Critical Code

	Applications
UNIX utilities	Coreutils, Busybox, Minix (over 450 apps)
UNIX file systems	ext2, ext3, JFS
Network servers	Bonjour, Avahi, udhcpd, lighttpd, etc.
Library code	libdwarf, libelf, PCRE, uClibc, etc.
Packet filters	FreeBSD BPF, Linux BPF
MINIX device drivers	pci, lance, sb16
Kernel code	HiStar kernel
Computer vision code	OpenCV (filter, remap, resize, etc.)
OpenCL code	Parboil, Bullet, OP2

- Most bugs fixed promptly

Coreutils Commands of Death

md5sum -c t1.txt	pr -e t2.txt
mkdir -Z a b	tac -r t3.txt t3.txt
mkfifo -Z a b	paste -d\\ abcdefghijklmnopqrstuvwxyz
mknod -Z a b p	ptx -F\\ abcdefghijklmnopqrstuvwxyz
seq -f %0 1	ptx x t4.txt
printf %d `	cut -c3-5,8000000- --output-d: file

t1.txt: \t \tMD5(

t2.txt: \b\b\b\b\b\b\b\t

t3.txt: \n

t4.txt: A

[Cadar, Dunbar, Engler OSDI 2008]
[Marinescu, Cadar ICSE 2012]

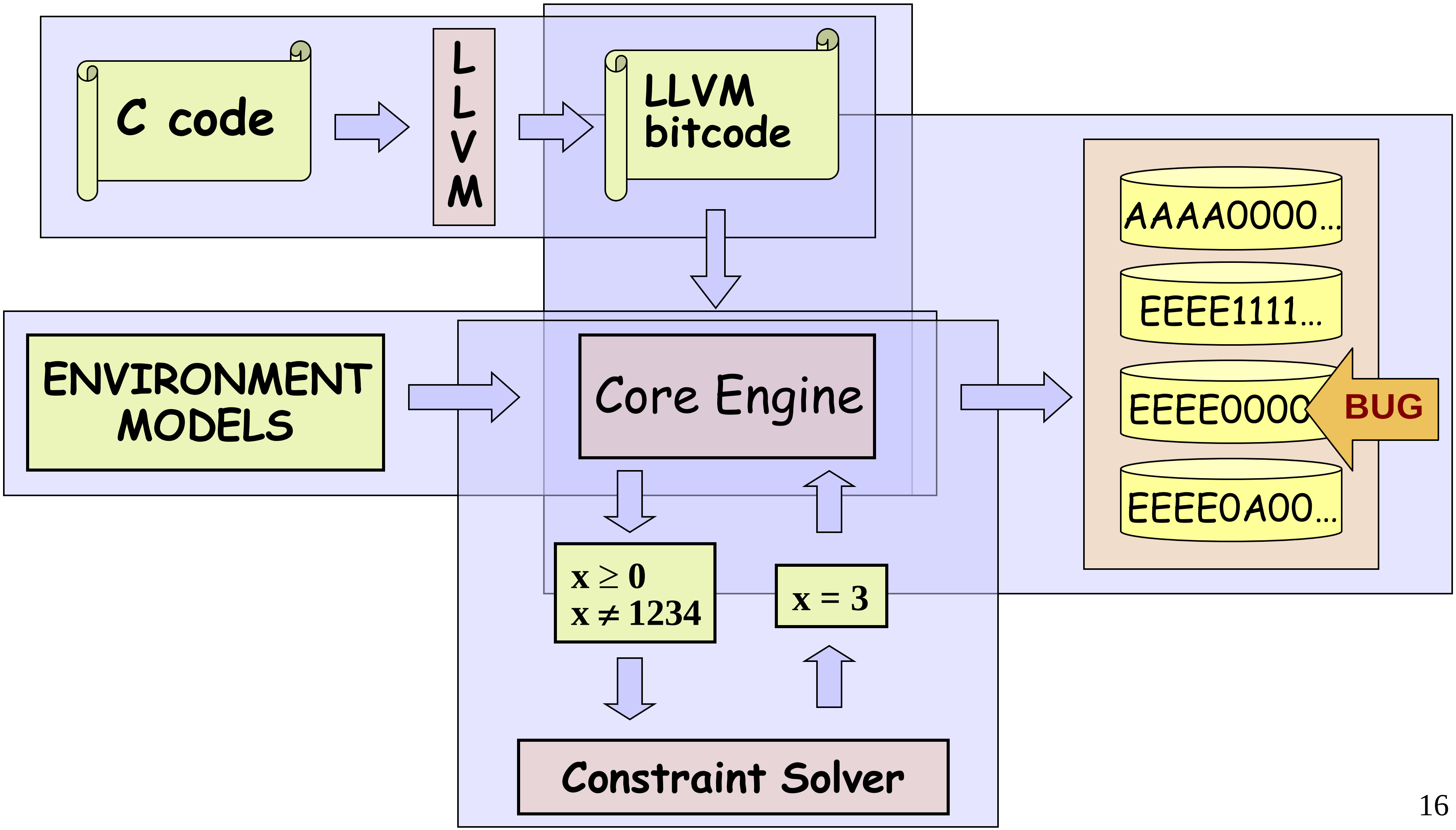
Packet of Death (Bonjour)

Offset	Hex Values							
0000	0000	0000	0000	0000	0000	0000	0000	0000
0010	003E	0000	4000	FF11	1BB2	7F00	0001	E000
0020	00FB	0000	14E9	002A	0000	0000	0000	0001
0030	0000	0000	0000	055F	6461	6170	045F	7463
0040	7005	6C6F	6361	6C00	000C	0001		

- **Causes Bonjour to abort, potential DoS attack**
- **Confirmed by Apple, security update released**

[Song, Cadar, Pietzuch IEEE TSE 2014]

KLEE Architecture



KLEE Demo: Toy Image Viewer

```
// #include directives
struct image_t {
  unsigned short magic;
  unsigned short h, sz; // height, size
  char pixels[1018];
};
int main(int argc, char** argv) {
  struct image_t img;
  int fd = open(argv[1], O_RDONLY);
  read(fd, &img, 1024);

  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  unsigned short w = img.sz / img.h;

  return w;
}
```

```
$ clang -emit-llvm -c -g image_viewer.c
$ klee --posix-runtime -write-pcs
  image_viewer.bc --sym-files 1 1024 A
...
KLEE: output directory = klee-out-1
(klee-last)
...
KLEE: ERROR: ... divide by zero
...
KLEE: done: generated tests = 4
```

KLEE Demo: Toy Image Viewer

```
$ cat klee-last/test000003.pc
...
array A-data[1024] : w32 -> w8 = symbolic
(query [
        ...
        (Eq  61166
            (ReadLSB w16 0 A-data))
        (Eq  0
            (ReadLSB w16 2 A-data))
        ...
)
```

KLEE Demo: Toy Image Viewer

```
$ klee-replay --create-files-only klee-last/test000003.ktest
[File A created]

$ xxd -g 1 -l 10 A
0000000: ee ee 00 00 00 00 00 00 00 00                  ..........

$ gcc -o image_viewer image_viewer.c
[image_viewer created]

$ ./image_viewer A
Floating point exception
```

KLEE Demo: All-Values Checks

```
int foo(unsigned k) {
    int a[4] = {3, 1, 0, 4};
    k = k % 4;
    return a[a[k]];
}

int main() {
    int k;
    klee_make_symbolic(&k,  sizeof(k), "k");
    return foo(k);
}
```

```
$ clang -emit-llvm -c -g all-values.c
$ klee all-values.bc
...
KLEE: ERROR: /home/klee/all-values/all-values.c:4: memory error: out of bound pointer
...
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2
```

Running KLEE inside a Docker container

Step 1: Install Docker for Linux/MacOS/Windows

Step 2: docker pull klee/klee

Step 3: docker run --rm -ti --ulimit='stack=-1:-1' klee/klee

http://klee.github.io/docker/

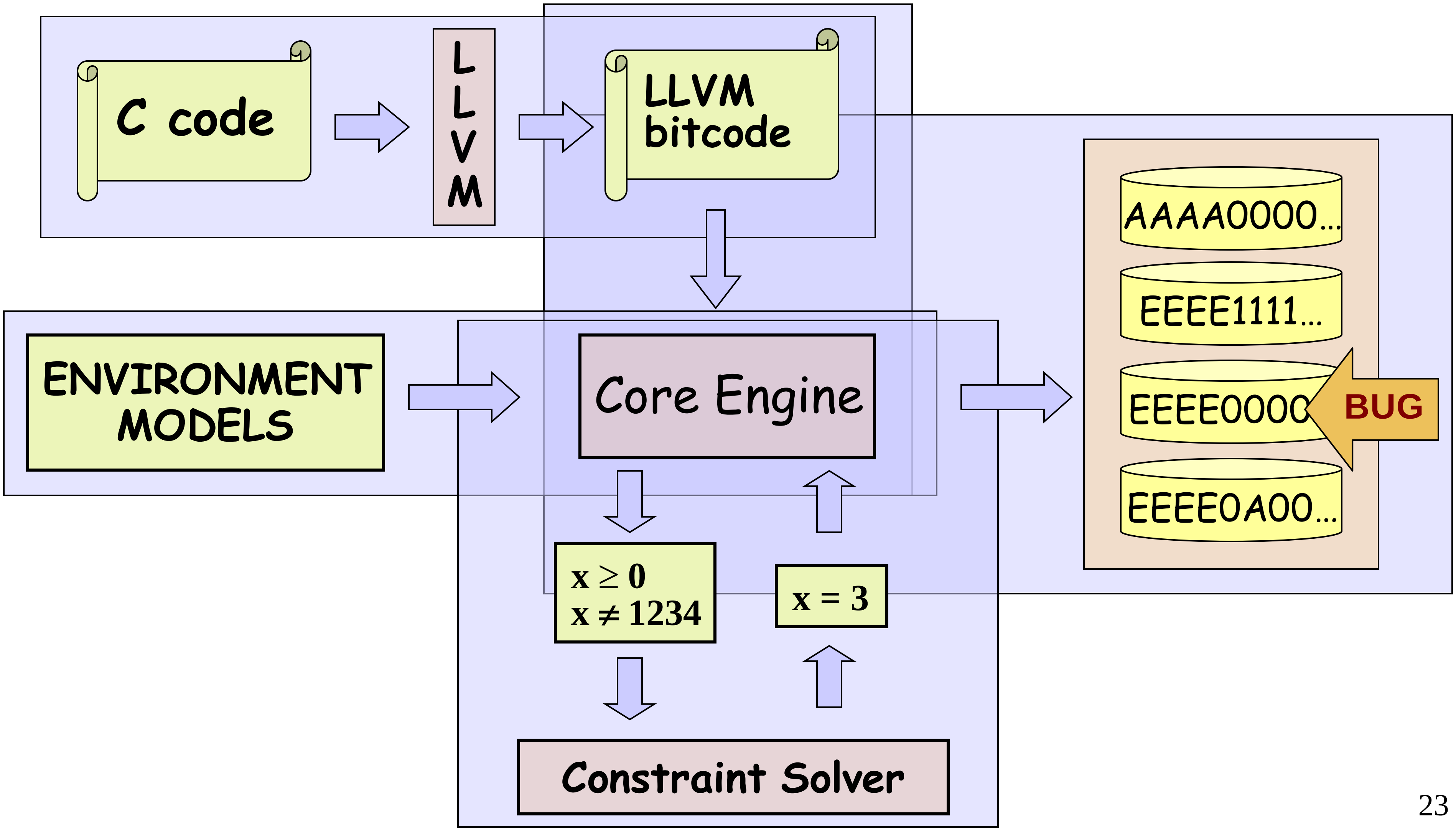
KLEE on the Web

You can try KLEE on the web now (world premiere!) at:

http://klee.doc.ic.ac.uk

(work in progress)

KLEE Architecture



KLEE Architecture:

LLVM

LLVM advantages:

- Mature framework, incorporated into commercial products by Apple, Google, Intel, etc.
- Elegant design patterns: analysis passes, visitors, etc.
- Single Static-Assignment (SSA) form with infinite registers (nice fit for symbolic execution)
- Lots of useful program analyses
- Well documented
- Several different front-ends, so KLEE could be extended to work with languages other than C

KLEE Architecture: L L V M

LLVM disadvantages

- Fast changing, not-backward compatible API!
  - KLEE is currently based on LLVM 3.4
- Compiling to LLVM bitcode is still not trivial, but it’s getting better:
  - make CC=“clang –emit-llvm”
  - LLVM Gold Plugin http://llvm.org/docs/GoldPlugin.html
  - Whole-Program LLVM https://github.com/travitch/whole-program-llvm

KLEE Architecture:

LLVM

KLEE runs LLVM, not C code!

```
#include <stdio.h>
int main() {
   int x;
   klee_make_symbolic(&x, sizeof(x), "x");

   if (x > 0)
       printf("x\n");
   else printf("x\n");

   return 0;
}
```

```
$ clang -emit-llvm -c -g code.c
$ klee code.bc


...
x

KLEE: done: total instructions = 6
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

KLEE Architecture:

Core Engine

The core engine implements symbolic execution exploration.

…	Interpreter	…
Memory	Core Engine	Stats
…	Searchers	…

KLEE Architecture:

Interpreter

Core Engine

- Works as a mixed concrete/symbolic interpreter for LLVM bitcode

```
Instruction *i = ki->inst;
 switch (i->getOpcode()) {
    case Instruction::Ret:
    …
    case Instruction::Br:
      // if both sides feasible, fork
      …
```

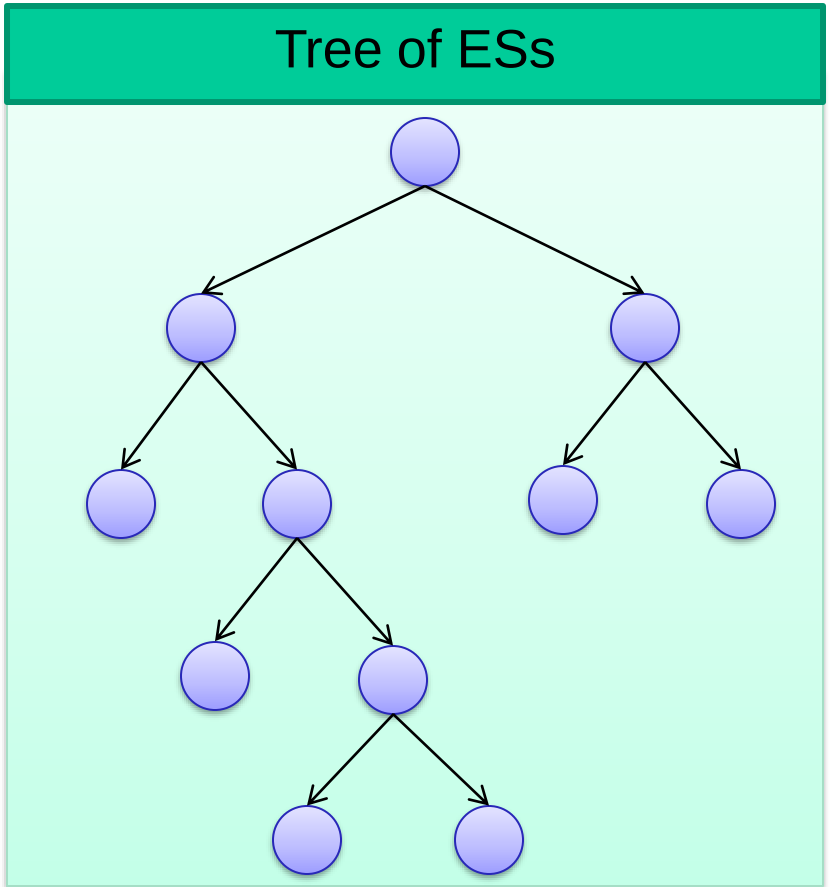
$ ./program ⟷ $ klee program.bc

Paths and Execution States

- Each path represented by an *ExecutionState*, with KLEE acting as an OS for ExecutionStates

ExecutionState

- PC
- Stack
- Address space
- List of sym objects
- Path constraints
- etc.



- Fork implemented by object-level COW

KLEE Architecture: Core Engine

The core engine implements symbolic execution exploration.

Two main **scalability challenges**:

Path exploration challenges

Constraint solving challenges

Path Exploration Challenges

Naïve exploration can easily get “stuck”

- Employing search heuristics
- Dynamically eliminating redundant paths
- Statically merging paths
- Using existing regression test suites to prioritize execution
- etc.

Search Heuristics in KLEE

Core Engine

Searchers

- Basic search heuristics such as BFS and DFS

  `klee --search=bfs program.bc`

- Coverage-optimized search (`--search=nurs:md2u`)
  - Select path closest to an uncovered instruction
- Random-state search (`--search=random-state`)
  - Randomly select a pending state/path
- Random-path search (`--search=random-path`)
  - Described next
- etc.

[Cadar, Ganesh, Pawlowski, Dill, Engler CCS'06]
[Cadar, Dunbar, Engler OSDI'08]
[Marinescu, Cadar ICSE'12], etc.

Random Path Selection

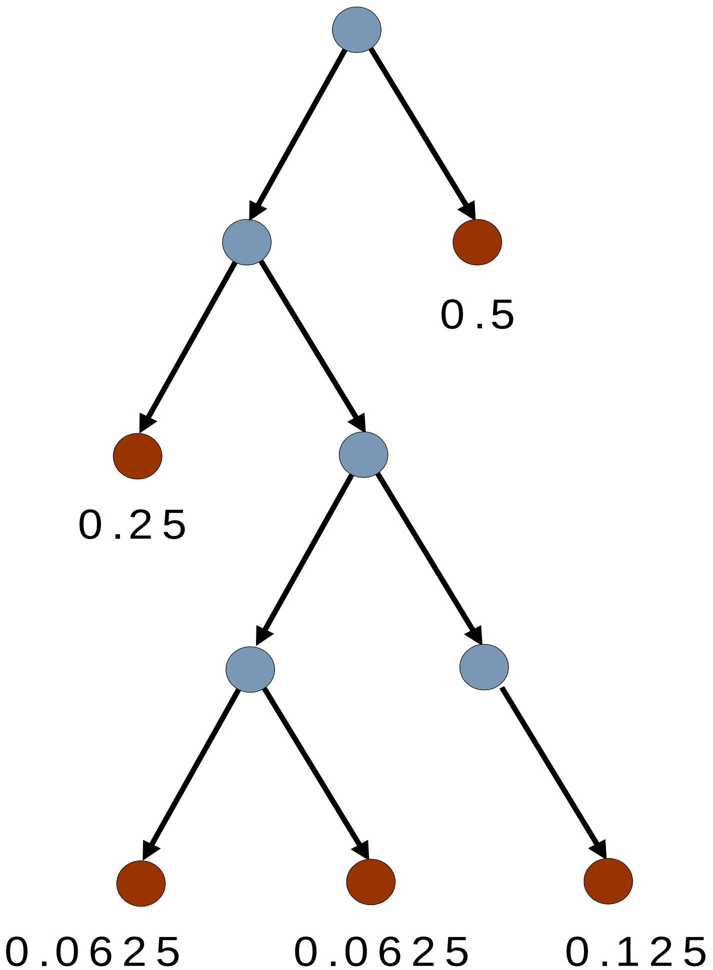
Core Engine

Searchers

- Maintain a binary tree of active paths
- Subtrees have equal prob. of being selected, irresp. of size

---

- NOT random state selection
- NOT BFS
- Favors paths high in the tree
  - fewer constraints
- Avoid starvation
  - e.g. symbolic loop



Combining Search Heuristics

KLEE can also use multiple heuristics in a round-robin fashion, to protect against individual heuristics getting stuck in a local maximum.

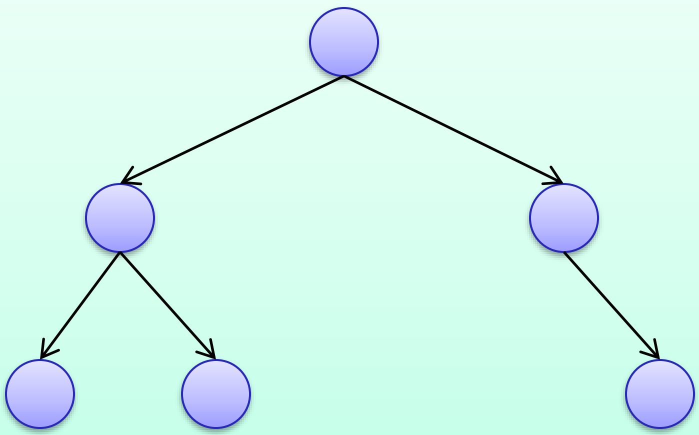
```
klee --search=nurs:md2u --search=dfs
     --search=random-path ...
```

New Search Heuristics

Core Engine

Searchers

Tree of ESs

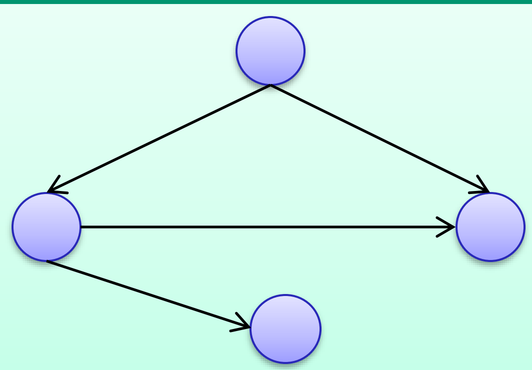


Easy to plug a new searcher by extending the Searcher class:

selectState() → ExecutionState

update(addedStates, removedStates)

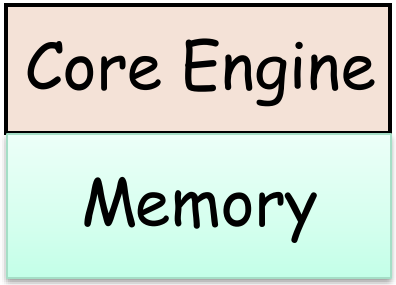
CFG



Statistics

- Solver time
- Instructions executed
- Memory consumption
- etc.

Memory Modelling



Accuracy: need bit-level modeling of memory:

- Systems code often observes the same bytes in different ways: e.g., using pointer casting to treat an array of chars as a network packet, inode, etc.
- Bugs (in systems code) are often triggered by corner cases related to pointer/integer casting and arithmetic overflows

Memory Modelling

Core Engine

Memory

- One data type: **arrays of bitvectors (BVs)**
- Mirror the (lack of) type system in C
  - Model each memory block as an array of 8-bit BVs
  - Bind types to expressions, not bits
- We can translate all C expressions into constraints in the theory of quantifier-free BV with arrays (QF_ABV) with bit-level accuracy
  - Main exception: floating-point

Accuracy: Example

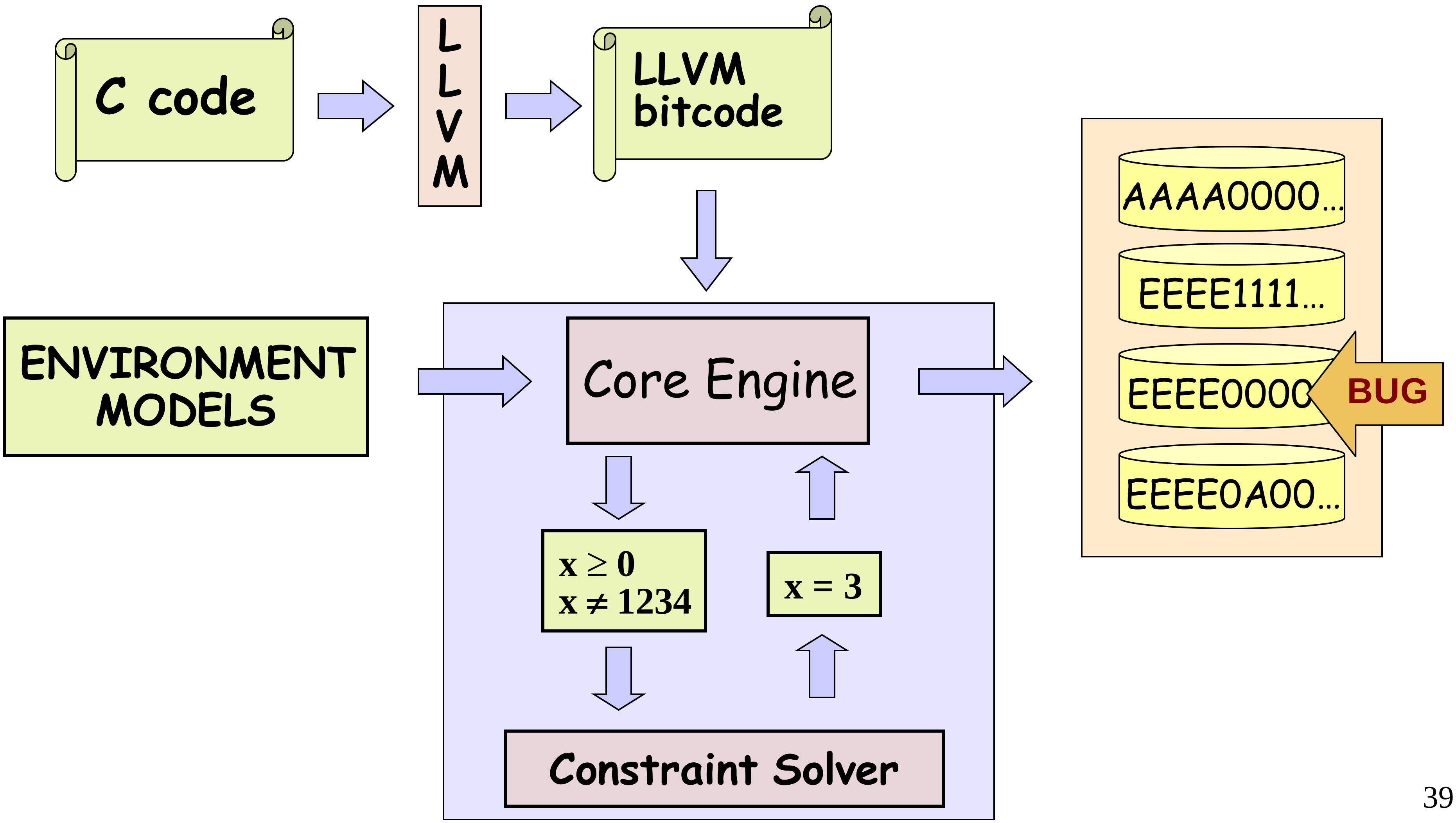
```
char buf[N]; // symbolic
struct pkt1 { char x, y, v, w; int z; } *pa = (struct pkt1*) buf;
struct pkt2 { unsigned i, j; } *pb = (struct pkt2*) buf;
if (pa[2].v < 0) { assert(pb[2].i >= 1<<23); }
```

buf: ARRAY BITVECTOR(32) OF BITVECTOR(8)

$\texttt{buf[18]} <_{\texttt{SIGNED}} \texttt{0x00}$

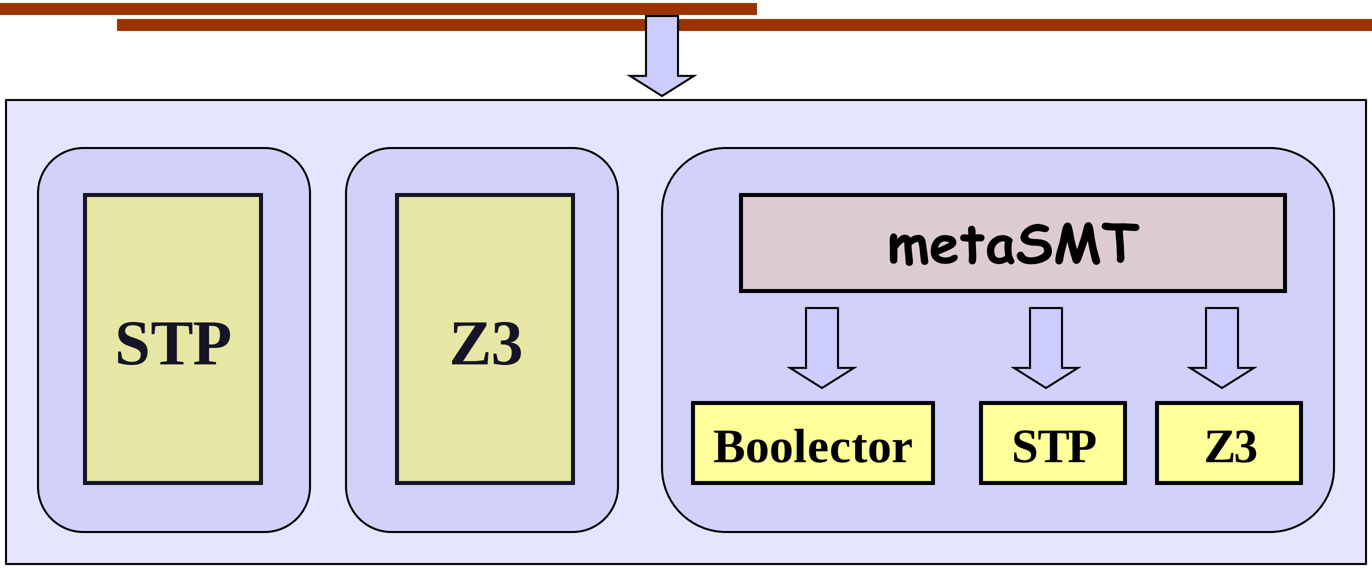
$\texttt{buf[19]@buf[18]@buf[17]@buf[16]} \geq_{\texttt{UNSIGNED}} \texttt{0x00800000}$

KLEE Architecture



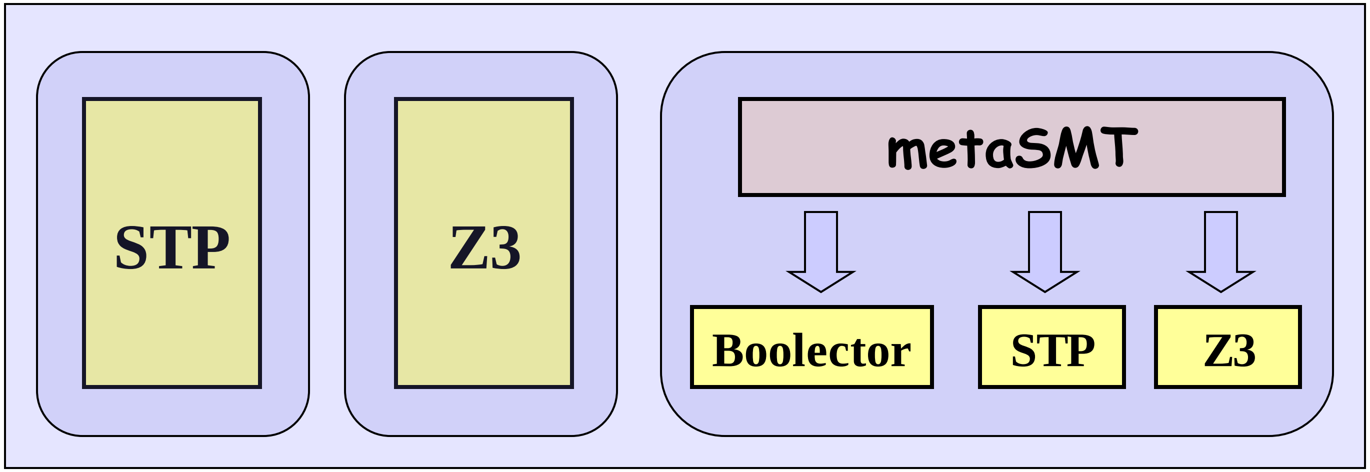
SMT Solvers
(--solver-backend=stp, z3, ...)

Theory of closed quantifier-free formulas over bitvectors and arrays of bitvectors (QF_ABV)



- **STP**: Developed at Stanford. Initially targeted to, and driven by, EXE. Main solver in KLEE.
- **Z3**: Developed at Microsoft Research, integrated both natively and as part of metaSMT.
- **Boolector**: Developed at Johannes Kepler University, integrated via metaSMT.

metaSMT

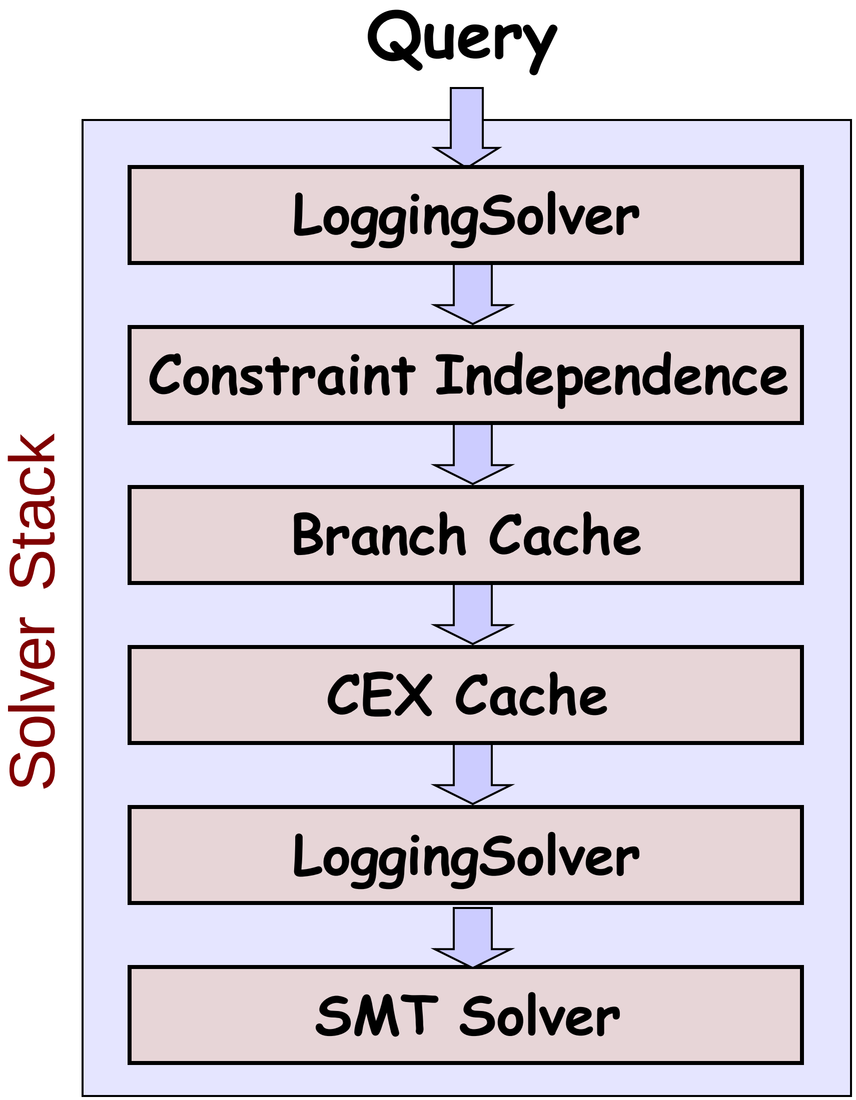


- **metaSMT** developed at University of Bremen provides a unified API for transparently using a number of SMT (and SAT) solvers
  - Avoids communication via text files, which would be too expensive
  - Small overhead: compile-time translation via metaprogramming

KLEE Architecture:

Constraint Solver

- Several high-level optimizations specific to symex
  - CEX caching, elimination of irrelevant constraints, etc.
- Implemented as a stack of solver passes
- Caching → only some queries reach the solver
- Independent **Kleaver** tool that implements this solver stack



Constraint Solving: Performance

- Inherently expensive
- Invoked at every branch

- Key insight: exploit the characteristics of constraints generated by symex

Some Constraint Solving Statistics
[after optimizations]

Application	Instrs/s	Queries/s	Solver %
[	695	7.9	97.8
base64	20,520	42.2	97.0
chmod	5,360	12.6	97.2
comm	222,113	305.0	88.4
csplit	19,132	63.5	98.3
dircolors	1,019,795	4,251.7	98.6
echo	52	4.5	98.8
env	13,246	26.3	97.2
factor	12,119	22.6	99.7
join	1,033,022	3,401.2	98.1
ln	2,986	24.5	97.0
mkdir	3,895	7.2	96.6
Avg:	**196,078**	**675.5**	**97.1**

1h runs using KLEE with DFS and no caching

UNIX utilities (and many other benchmarks)

- Large number of queries
- Most queries <0.1s
- Most time spent in the solver (before and after optimizations!)

[Palikareva and Cadar CAV'13]

Higher-Level Constraint Solving Optimizations

- Two simple and effective optimizations
  - Eliminating irrelevant constraints
  - Caching solutions

Eliminating Irrelevant Constraints

(--use-independent-solver=true/false)

- In practice, each branch usually depends on a small number of variables

```
…
…
if (x < 10) {
    …
}
```

~~**w+z > 100**~~

~~**2 * w – 1 < 12345**~~

x + y > 10

~~**z & -z = z**~~

x < 10 ?

[CCS'06]

Caching Solutions

(--use-cex-cache=true/false)

- Static set of branches: lots of similar constraint sets

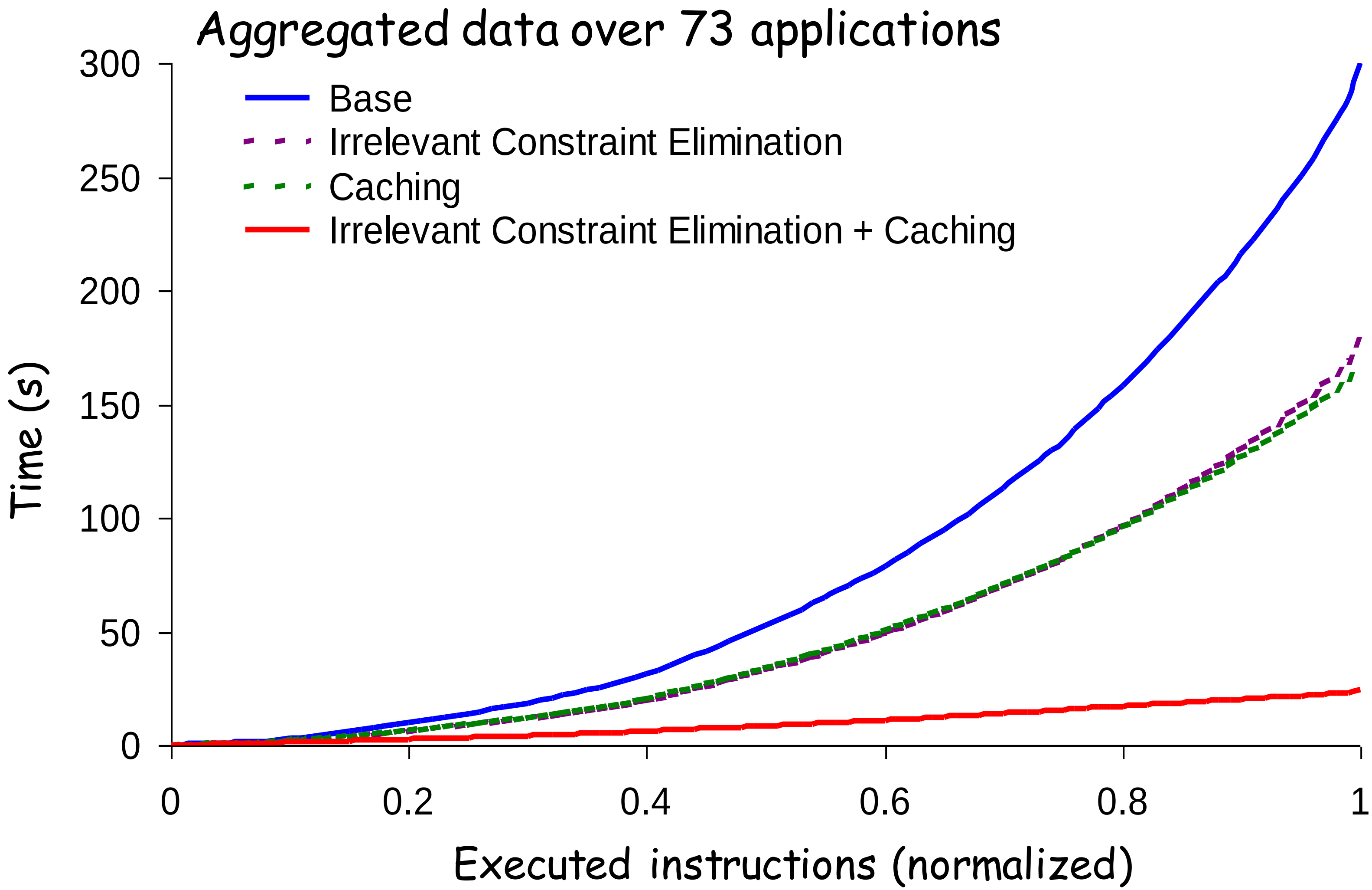
$2 * y < 100$
$x > 3$
$x + y > 10$
→
$x = 5$
$y = 15$

$2 * y < 100$
$x + y > 10$
→ Eliminating constraints cannot invalidate solution →
$x = 5$
$y = 15$

$2 * y < 100$
$x > 3$
$x + y > 10$
$x < 10$
→ Adding constraints often does not invalidate solution →
$x = 5$
$y = 15$

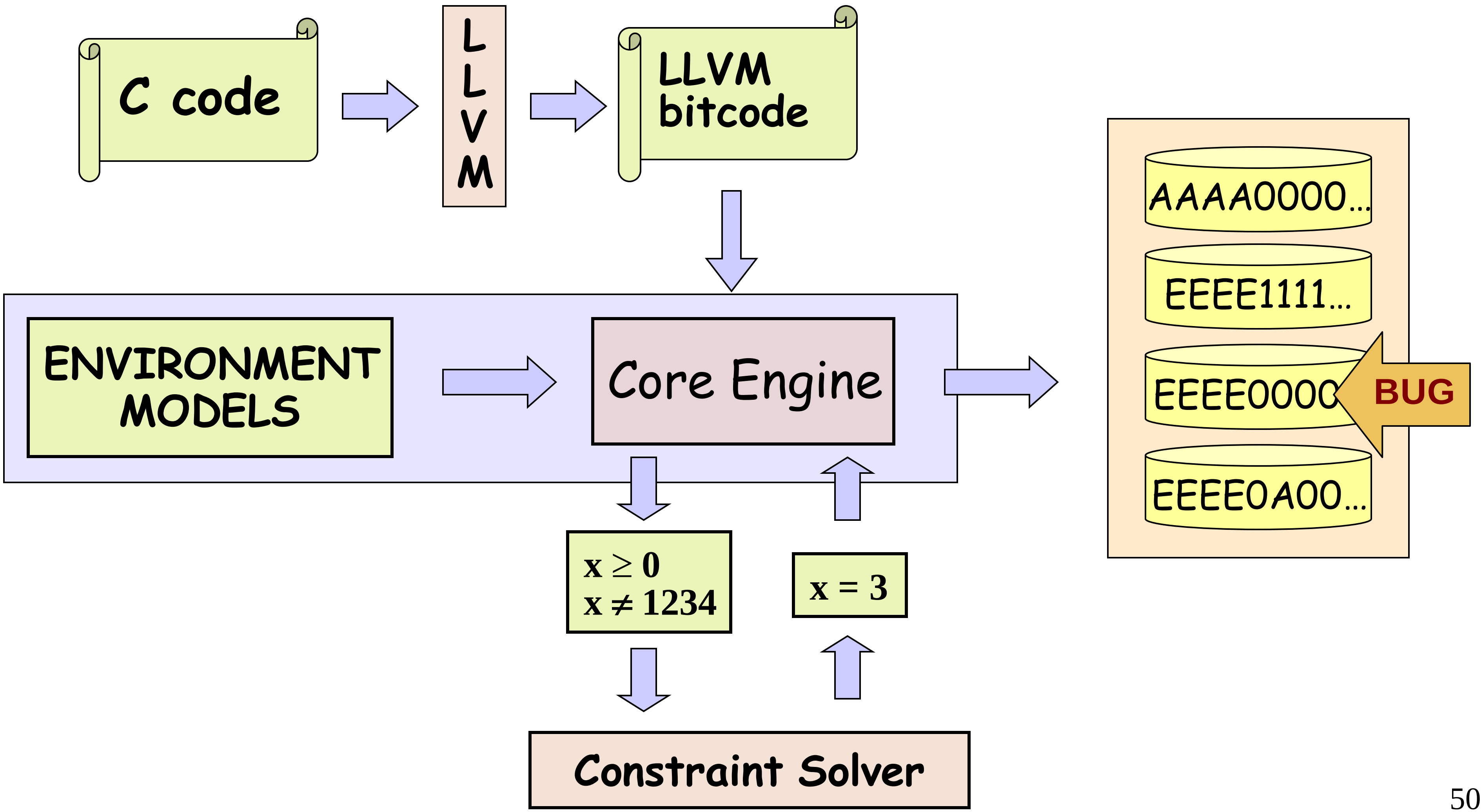
[OSDI'08]

Speedup



Aggregated data over 73 applications
Time (s)
300
250
200
150
100
50
0
Base
Irrelevant Constraint Elimination
Caching
Irrelevant Constraint Elimination + Caching
0
0.2
0.4
0.6
0.8
1
Executed instructions (normalized)

KLEE Architecture



KLEE Architecture:

Environment Models

- Environment model: model for a piece of code for which source is not available
- In KLEE, the environment is mainly the OS system call API

Environmental Modeling

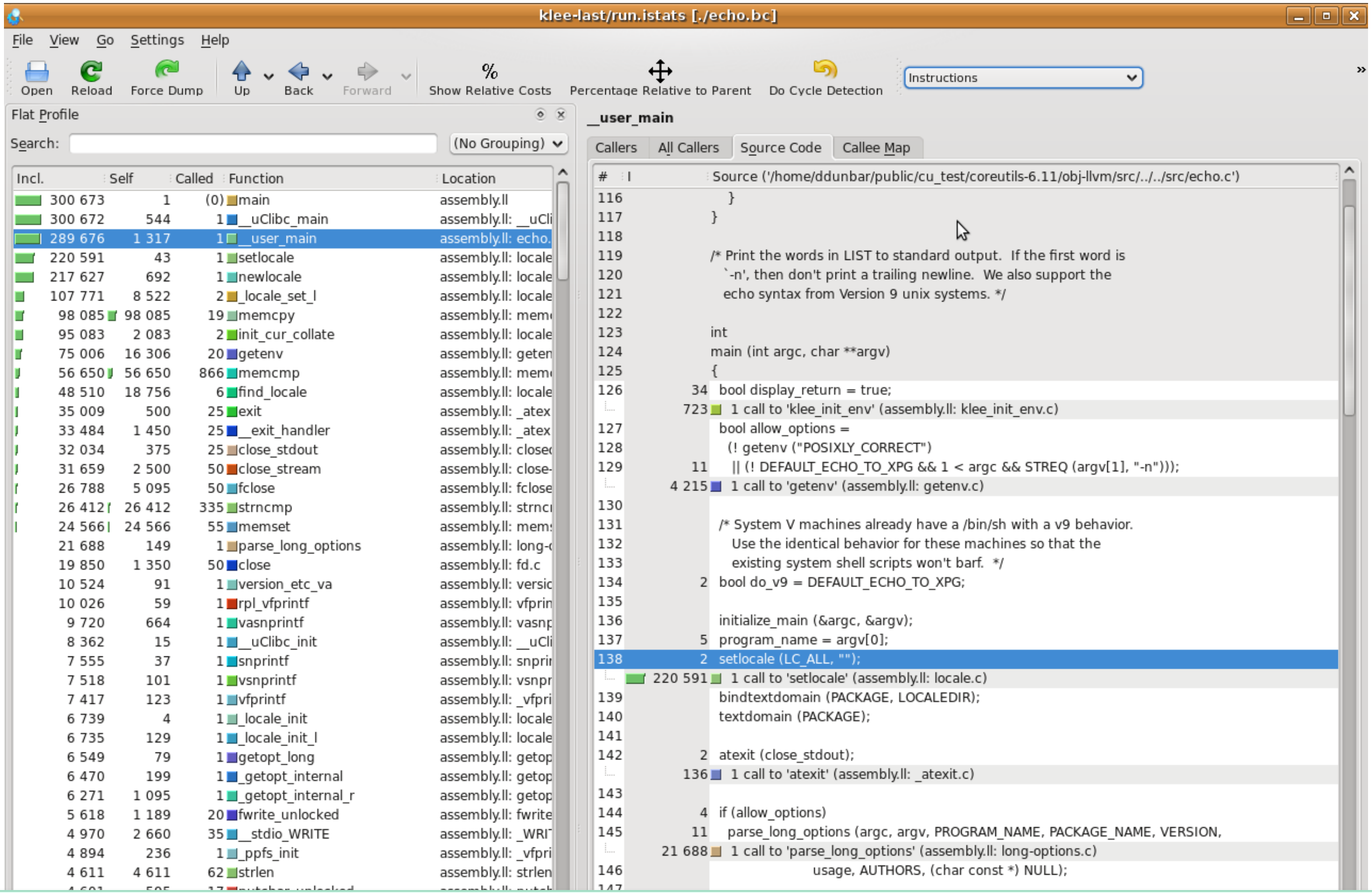
Models are plain C code, which KLEE interprets as any other code!

```
// actual implementation: ~50 LOC
ssize_t read(int fd, void *buf, size_t count) {
        klee_file_t *f = get_file(fd);
        …
        memcpy(buf, f->contents + f->off, count)
        f->off += count;
        …
```

- Users can extend/replace environment w/o any knowledge of KLEE's internals
  - Often the first part of KLEE users experiment with
- Users can choose precision
  - fail system calls? etc.
- Currently: effective support for symbolic command line arguments, files, links, pipes, ttys, environment vars

Statistics

Core Engine | Stats



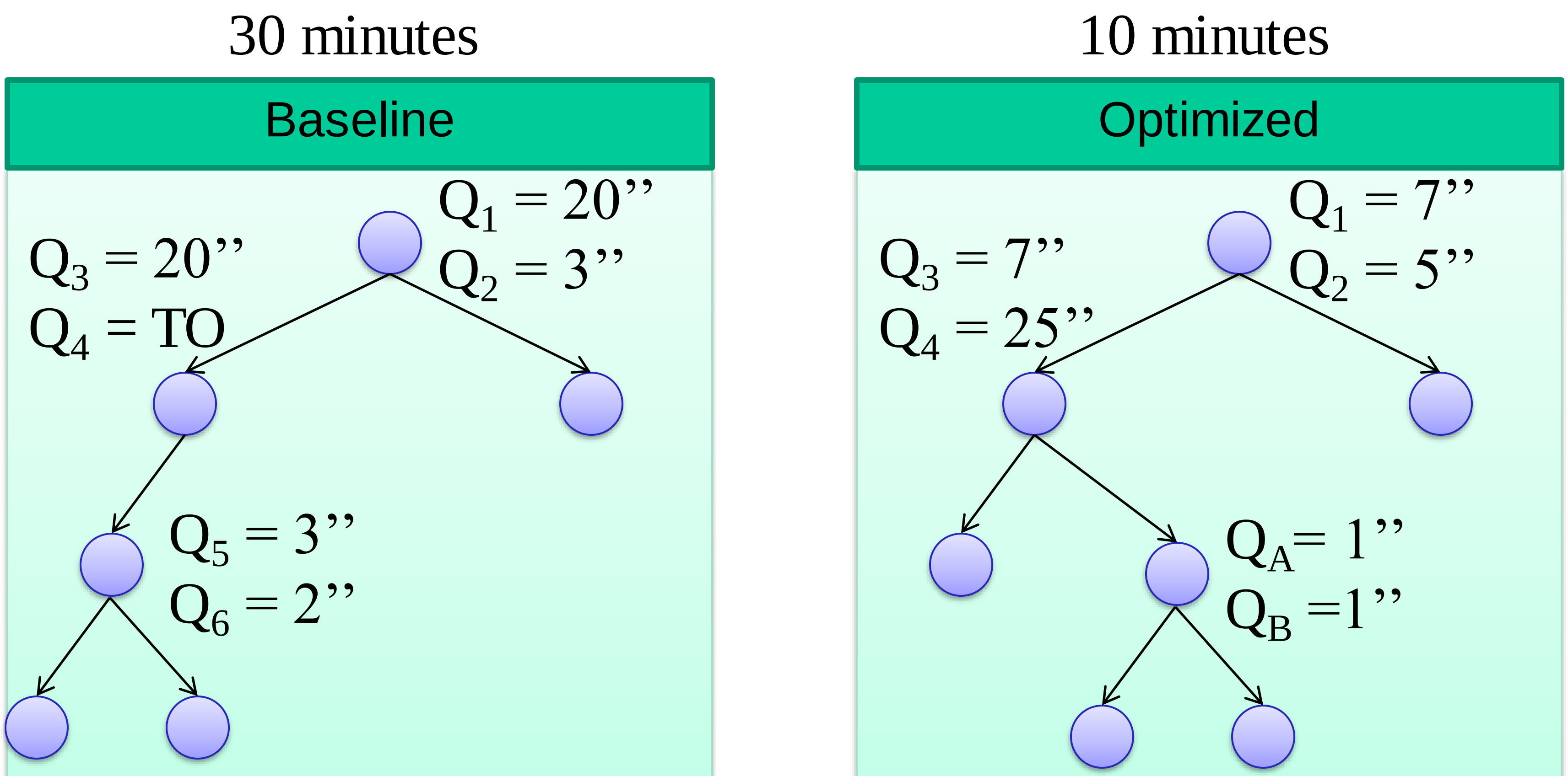
Good support for producing and visualizing a variety of statistics, associated with different entities and events

Non-determinism in SymEx and KLEE

- Any good experiment needs to take non-determinism into account
- Sources of non-determinism include constraint solving, search heuristics, LLVM versions, memory allocation
  - Currently fixing implementation-level non-determinism, such as hash tables indexed by memory addresses, which can differ across runs

Example: Constraint solving optimization in KLEE

Approach: run baseline KLEE for 30', rerun in the same configuration with optimizations



Example 2: Coverage optimization in KLEE

Approach: take same benchmarks from paper X, rerun KLEE with coverage optimization

Baseline (LLVM 2.3)	Baseline (LLVM 3.4)	Optimized (LLVM 3.4)
60% coverage	80% coverage	80% coverage

Dynamic Symbolic Execution

- Program analysis technique that can be use to automatically explore paths through a program
- Can generate inputs achieving high-coverage and exposing bugs in complex software

KLEE: Freely Available as Open-Source

http://klee.github.io/

- Popular symbolic execution tool with an active user and developer base
- Extended in many interesting ways by several groups from academia and industry, in areas such as:
  - exploit generation
  - wireless sensor networks/distributed systems
  - automated debugging
  - client-behavior verification in online gaming
  - GPU testing and verification
  - etc. etc.